Has the Bug Really Been Fixed?[*] [†]

Zhongxian Gu Earl T. Barr David J. Hamilton Zhendong Su
Department of Computer Science, University of California at Davis
{zgu,etbarr,davidjh,su}@ucdavis.edu

ABSTRACT

Software has bugs, and fixing those bugs pervades the software engineering process. It is folklore that bug fixes are often buggy themselves, resulting in *bad fixes*, either failing to fix a bug or creating new bugs. To confirm this folklore, we explored bug databases of the Ant, AspectJ, and Rhino projects, and found that bad fixes comprise as much as 9% of all bugs. Thus, detecting and correcting bad fixes is important for improving the quality and reliability of software. However, no prior work has systematically considered this *bad fix problem*, which this paper introduces and formalizes. In particular, the paper formalizes two criteria to determine whether a fix resolves a bug: *coverage* and *disruption*. The coverage of a fix measures the extent to which the fix correctly handles all inputs that may trigger a bug, while disruption measures the deviations from the program's intended behavior after the application of a fix. This paper also introduces a novel notion of *distance-bounded weakest precondition* as the basis for the developed practical techniques to compute the coverage and disruption of a fix.

To validate our approach, we implemented FIXATION, a prototype that automatically detects bad fixes for Java programs. When it detects a bad fix, FIXATION returns an input that still triggers the bug or reports a newly introduced bug. Programmers can then use that bug-triggering input to refine or reformulate their fix. We manually extracted fixes drawn from real-world projects and evaluated FIXATION against them: FIXATION successfully detected the extracted bad fixes.

[*]This research was supported in part by NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, NSF CCF Grant No. 0702622, and the U.S. Air Force under grant FA9550-07-1-0532. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

[†]This material is based in part upon work supported by the U.S. Department of Homeland Security under Grant Award Number 2006-CS-001-000001, under the auspices of the Institute for Information Infrastructure Protection (I3P) research program. The I3P is managed by Dartmouth College. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the U.S. Department of Homeland Security, the I3P, or Dartmouth College.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; F.3.1 [**Theory of Computation**]: Specifying and Verifying and Reasoning about Programs

General Terms

Languages, Reliability, Verification

Keywords

Bug fixes, Weakest precondition, Symbolic execution

1. INTRODUCTION

[**?**] According to IDC [21], software maintenance cost $86 billion in 2005, accounting for as much as two-thirds of the overall cost of software production. Developers spend 50–80% of their time looking for, understanding, and fixing bugs [7]. Fixing bugs correctly the first time they are encountered will save money and improve software quality.

Researchers have paid great attention to detecting and classifying bugs to ease developers' work [2, 9, 12, 13, 14, 18, 30, 35, 37]. In contrast, not much effort has been expended on bug fixes. When testing a fix, programmers usually rerun a program with the bug-triggering input. If no error occurs, programmers, hurried and overburdened as they usually are, often move on to their next task, thinking they have fixed the bug. Folklore suggests that bug fixes are frequently bad, either by failing to *cover*, *i.e.* handle, all bug-triggering inputs, or by introducing *disruptions*, *i.e.* new bugs. A bad fix can decrease the quality of a program by concealing the original bug, rendering it more subtle and increasing the cost of its fix. Thus, detecting and correcting bad fixes as soon as possible is important for the quality and reliability of software.

To gain insight into the prevalence and characteristics of bad fixes, we explored the Bugzilla databases [5] of the Ant, AspectJ and Rhino projects under Apache, Eclipse and Mozilla foundations. At the time of our survey, these databases contained entries to July 2009. In Bugzilla, a bug is *reopened* if it fails regression testing or a symptom of the bug recurs in the field. We hypothesized that a reopened bug might indicate a bad fix. We read the comment histories of reopened bugs to judge whether or not the bug was reopened due to a bad fix. Programmers sometimes even admit they committed a bad fix: in our survey, we found statements like "Oh, I am sorry I didn't consider that possibility" and "Oops, missed one code path." Of all reopened bugs, we found that bad fixes account for 66% in Ant, 73% in AspectJ, and 80% in Rhino. Bugs reopened because they failed a regression test belong to the disruption dimension of a bad fix. In our preliminary findings, reopened bugs comprise 4–7.25% of *all* bugs in these three projects[1]. We also found that 38–50% of bugs reopened due to a bad fix either had duplicates or blocked other bugs and were therefore linked to other bugs in Bugzilla. We found a total of 1,977 bad fixes from reopened bugs[2], and an additional 830 duplicates (*i.e.* bugs marked as a duplicate of a bad fix), comprising 9% of the total bugs (31,201) in the Apache database. Bad fixes need not manifest in reopened bugs; we focused on reopened bugs because they often make bad fixes easier to identify. Our manual study undercounts the prevalence of bad fixes in the studied projects, although we cannot say by how much.

In this paper, we describe the first systematic treatment of the bad fix problem: we introduce and formalize the problem, and present novel techniques to help developers assess the quality of a bug fix. We deem a fix bad if it fails to cover all bug-triggering inputs or introduces new bugs. An ideal fix covers all bug-triggering inputs and introduces no new bugs. We define two criteria to determine whether or not a fix resolves a bug — coverage and disruption:

Coverage: Many inputs may trigger a bug. The *coverage* of a fix measures the extent to which the fix correctly handles all bug-triggering inputs.

Disruption: A fix may unexpectedly change the behavior of the original program. *Disruption* counts these deviations from the program's intended behavior introduced by a fix.

Given a buggy program, a bug-triggering input that results in an assertion failure, a test-suite, and a fix, the *bad fix problem* is to determine the coverage and disruption of the fix.

In theory, Dijkstra's weakest precondition (WP) [11] can be used to calculate the coverage of a fix. We start from the manifestation of the bug in the buggy version of the program to discover the set predicate of bug-triggering subset of the program's input domain. Then we would symbolically execute the fixed program to learn whether, starting from that set predicate, inputs still exist that can trigger the bug. However, Dijkstra's WP computation depends on loop invariants and must contend with paths exponential in the number of branches.

We propose *distance-bounded weakest precondition (WP_d)* to perform the WP calculation over a set of paths near a concrete path. Given a path and a distance budget, WP_d produces a set of paths and computes the disjunction of the WP of each path. In the context of the bad fix problem, the bug-triggering input induces this concrete path. This process is sound and yields an under-approximation of the set of bug-triggering inputs. We can improve our under-approximation by increasing the distance budget. Indeed, in the limit when the distance budget tends to infinity, WP_d is precisely Dijkstra's WP.

WP_d offers two practical benefits. First, because we operate directly on paths, we avoid both the loop invariant requirement and the path-explosion problem. Second, it is our intuition that paths closer to the bug-triggering path are more likely to be related to the same bug and more error-prone, so adding them is likely to quickly approximate the bug-triggering input domain at low cost. Kim *et al.* showed that when a bug occurs in a file, more bugs are likely to occur in that file, a phenomenon they name temporal locality [23]. Their results can be put another way: defects are lexically clustered, which supports our intuition since many execution paths that are close to each other are also lexically close.

[1]The absolute numbers are $\frac{377}{5200}$ for Ant, $\frac{86}{2162}$ for AspectJ, and $\frac{38}{847}$ for Rhino. The number is $\frac{2939}{31201}$ (9.4%) across all Apache projects.

[2]Here we restricted ourselves to bugs that had been reopened, but not marked as duplicates.

This approach may appear circuitous: why not apply WP_d directly to the fixed program to see if we can find an input that triggers the assertion failure? The problem is that the original buggy input no longer triggers the bug in the fixed program. Thus, the concrete path that triggers the bug in the buggy version of the program no longer reaches the assertion and may not even exist in fixed program. Computing WP based on this false path may lead to spurious or incorrect results.

Regression testing is a measure of our disruption criterion; a project's test suite is a parameter of the bad fix problem to take advantage of this fact. We combine random and regression testing to calculate the disruption of a fix.

To demonstrate the feasibility of our approach, we implemented a prototype, FIXATION, which automatically detects bad fixes in Java programs. Given the buggy and fixed, versions of a program, a test-suite, and a bug-triggering input, FIXATION solves the bad fix problem. Our tool currently supports Java programs with conditionals in Boolean and integer domains. When it detects a coverage failure, it outputs a counterexample that triggers the bug in the fixed program (See Section 3.3); when it detects a disruptive fix, it reports the failing test cases or inputs (See Section 3.4). From examining the counterexample or the failing test cases, programmer can understand why the fix did not work and improve it.

The main contributions of this paper are:

- We introduce the bad fix problem and provide empirical evidence of its importance by exploring the bug databases of three real projects to find that bad fixes accounts for as much as 9% of all bugs.
- We formalize the bad fix problem and propose distance-bounded weakest precondition (WP_d), a novel form of weakest precondition, well-suited for the bad fix problem, that restricts the weakest precondition computation to a subset of the paths in a program's control flow graph.
- We implemented a prototype, called FIXATION, to check the coverage and disruption of a fix. We evaluated our prototype to demonstrate the feasibility of our approach: FIXATION detects bad fixes extracted from real-world programs.

The structure of this paper is as follows. In Section 2, we illustrate the problem with actual bad fixes and show how our technique can detect them. Section 3 formalizes our criteria for a bad fix and presents the detailed technique to measure them. We describe our prototype implementation and evaluation results in Section 4. Finally, we survey related work (Section 5) and conclude the paper with a discussion of future work (Section 6).

2. ILLUSTRATIVE EXAMPLE

This section describes an actual sequence of bad fixes for a bug from the Rhino project, and how our approach would have helped.

Rhino is an open-source JavaScript interpreter written in Java. JavaScript allows programmers to define `_noSuchMethod_`, a special method that the JavaScript interpreter invokes, instead of raising an exception, when an undefined method is called on an object. The bug, which we name `NoSuchMethod`, was a lack of support for this `_noSuchMethod_` mechanism. Its fix is not complicated; the final patch was less than 100 lines. However, due to bad fixes, the bug was reopened twice and three fixes were committed in three months.

Figure 1 contains the first committed fix. On Line 1, the programmer admits that he was not sure whether this fix covered all relevant inputs. The fix adds an `if`-block which, when an undefined method has been called, extracts `noSuchMethodMethod`

```
// no idea what to do if it's a TAIL_CALL
if ( fun instanceof NoSuchMethodShim
     && op != Icode_TAIL_CALL){

  // get the shim and the actual method
  NoSuchMethodShim =(NoSuchMethodShim)fun;
  Callable noSuchMethodMethod =
      noSuchMethodShim.noSuchMethodMethod;
  ...
}
```

Figure 1: First fix of `NoSuchMethod`.

```
if ( fun instanceof NoSuchMethodShim ) {
~~if ( fun instanceof NoSuchMethodShim~~
    ~~&& op != Icode_TAIL_CALL) {~~

    // get the shim and the actual method
    NoSuchMethodShim = (NoSuchMethodShim)fun;
    Callable noSuchMethodMethod =
        noSuchMethodShim.noSuchMethodMethod;
    ...
    if ( op == Icode_TAIL_CALL ) {
        callParentFrame = frame.parentFrame;
        exitFrame(cx, frame, null);
    }
    ...
}
```

Figure 2: Second fix of `NoSuchMethod`. Green, normal weight lines indicate changes added in this fix; red, strikeout lines indicate those removed; and gray lines are those left unchanged.

from `NoSuchMethodShim` and dispatches the undefined method on it, passing the original test case. However, the clause "`op != Icode_TAIL_CALL`" could be false for an undefined method call. The programmer missed this case. Under our criteria, this fix fails the coverage check. Given buggy and fixed versions of the program, FIXATION would compute the predicate of the bug-triggering input domain and symbolically execute the fixed program with that predicate as the initial precondition. Upon reaching the exception, FIXATION would determine the fix to be bad, and return the counterexample "`fun instanceof NoSuchMethodShim` $\wedge$ `op == Icode_TAIL_CALL`" to the programmer. In this example, FIXATION can exploit the common idiom of asserting `false` at a code path not expected to be reached; in general, however, the assertion that captures a bug can be more complex.

After the bug was reopened, the programmer refined the fix, as shown in Figure 2. The clause that restricted the operation mode was dropped. Inside this `if`-block, the programmer added a block to deal with the case when operation mode was set to `Icode_TAIL_CALL`. The fix handles all inputs that triggered the original bug. However, the fix failed when subjected to regression testing. It fails the disruption check of a bad fix: it excised the bug that motivated its application at the cost of introducing new bugs.

Finally, the programmer committed a third version of the fix, which resolved the bug and passed the regression tests. This sequence of fixes shows how easy it is to write a bad fix. Programmers considering only of a subset of the bug-triggering input domain are likely to miss conditions and execution paths. Our technique can help programmers detect these conditions earlier and write better fixes more quickly.

3. APPROACH

To begin, we formalize our problem domain, then define the coverage and disruption of a fix. We abstract the bug b as a failure of the assertion φ. Ideally, we would directly compute whether Dijkstra's WP from φ in the fixed program is false. This computation requires loop invariants and must contend with the path-explosion problem. We have a bug-triggering input and its induced concrete failing path. The key idea of our approach for computing whether a fix covers all inputs that can trigger a bug is to leverage that failing path to compute a sound under-approximation of the bug-triggering input domain, then test the fixed program against inputs from that domain. In Section 3.2, we introduce WP$_d$ to compute that sound under-approximation. Section 3.3 shows how we use that under-approximation to symbolically execute the fixed program to calculate a counterexample to the fix. We close, in Section 3.4, by presenting our algorithm for computing fix disruption which combines regression and random testing.

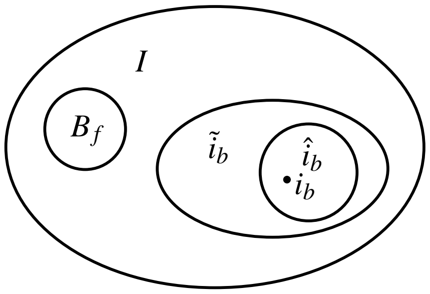


Figure 3: A program's input domain I, the known input i_b that triggers the bug b, all inputs that trigger the bug $\tilde{i}_b$, those inputs the fix f handles $\hat{i}_b$, new bugs B_f that f may introduce, and their inter-relationships.

3.1 The Bad Fix Problem

A good bug fix eliminates the bug for all inputs without introducing new bugs. We define two criteria to measure these dimensions — *coverage* and *disruption*.

We model a program $P : I \rightarrow O$ as a function in terms of its input/output behavior. We assume that the bug b causes an assertion failure in the buggy program P_b and that we know a bug-triggering input i_b such that Equation 1 holds. Concretely, i_b represents a failing test case, *i.e.*, the output $P_b(i_b)$ violates the assertion φ.

$$P_b(i_b) \not\models \varphi \quad \text{(or equivalently, } P_b(i_b) \models \neg\varphi) \tag{1}$$

Many inputs may trigger $\neg\varphi$. Definition 3.1 specifies this set.

DEFINITION 3.1 (BUG-TRIGGERING INPUT DOMAIN).

$$\tilde{i}_b = \{i \in I : P_b(i) \models \neg\varphi\}$$

A bug fix f creates a new version of the program P_f. At the very least, $P_f(i_b) \models \varphi$, but P_f may not handle all of $\tilde{i}_b$. Definition 3.2 defines the subset of $\tilde{i}_b$ that P_f handles.

DEFINITION 3.2 (COVERED BUG-TRIGGERING INPUTS).

$$\hat{i}_b = \{i \in \tilde{i}_b : P_f(i) \models \varphi\}$$

Ideally, a fix f eliminates the bug b and covers all of $\tilde{i}_b$. With respect to $\tilde{i}_b$, the set $\hat{i}_b$ indicates the degree to which a fix achieves this goal, *viz.* the first dimension of fix quality, its *coverage*. We use $cov(f)$ to denote the coverage of a fix f.

By definition, P_f is correct, relative to b, for the inputs in $\hat{i}_b$. Outside of $\tilde{i}_b$, a bad fix f may introduce new bugs. Definition 3.3 captures these bugs.

DEFINITION 3.3 (INTRODUCED BUGS). *Let P^o be the correct oracle for P,* i.e.*, for all inputs, P^o produces the desired output.*

$$B_f = \{i \in I \setminus \tilde{i}_b : P_f(i) \neq P^o(i)\} \tag{2}$$

The *disruption* of the fix f is the set of new deviating input values it introduces, *viz.* the set B_f. When f introduces no new bugs, $B_f = \emptyset$ and f is not disruptive. Since we do not have a program oracle in general, we approximate P^o with the program's test suite T and P_b, the buggy version of the program. Section 3.4 presents the algorithm we use to compute disruption. Along the disruption dimension, we compare the quality of two fixes in terms of the B_f sets they induce.

An *ideal fix* covers all bug-triggering inputs and introduces no disruptions:

$$\hat{i}_b = \tilde{i}_b \wedge B_f = \emptyset. \tag{3}$$

Figure 3 illustrates the interrelations of the sets defined in this section. With our coverage and disruption properties in hand, we now define the bad fix problem.

DEFINITION 3.4 (BAD FIX PROBLEM). *Given a buggy program P_b, a bug-triggering input i_b, a test suite $T : I \to O$ (modeled as a* partial function *from I to O), and the fix f, determine the coverage and disruption of f.*

We can also use our criteria to partially order fixes for the same bug. The fix f_a is better than f_b if and only if

$$cov(f_b) \subseteq cov(f_a) \wedge B_{f_a} \subseteq B_{f_b}. \tag{4}$$

3.2 Distance-Bounded Weakest Precondition

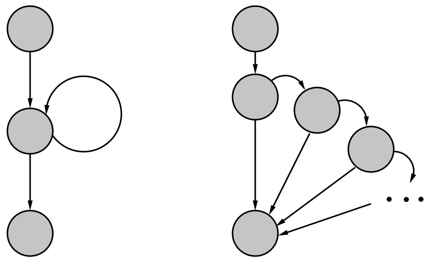
To determine whether a fix covers the bug-triggering, we introduce the concept of distance-bounded weakest precondition (WP$_d$) which generalizes Dijkstra's weakest precondition (WP). WP$_d$ restricts the weakest precondition computation to a subset of the paths near a distinguished path in the interprocedural control flow graph (ICFG) of a program. In the context of the bad fix problem, the distinguished path is Π_{i_b}, the concrete path induced by the known bug-triggering input i_b. In this section, we explain how WP$_d$ traverses the ICFG of a program and uses Levenshtein edit distance [25] to construct the subset of simple paths over which WP$_d$ computes the weakest precondition. By considering only a subset of simple paths, WP$_d$ mitigates the path-explosion problem and does not need loop invariants:

$$\text{WP}_d : \text{Programs} \times \text{Predicates} \times \text{Paths} \times \mathbb{N}_0 \to \text{Predicates}. \tag{5}$$

Equation 5 defines the signature of WP$_d$, which adds a path Π and a distance d to the signature of standard WP. An application of WP$_d(P, \varphi, \Pi, d)$ first generates the set C of candidate paths at most d distance from Π, then computes the standard weakest precondition over only the candidate paths in C. Equation 6 defines the candidate paths WP$_d$ considers. The metric Δ computes the distance of two paths. Currently, we assign a symbol to every edge in the program's ICFG, map every path to a string, and use Levenshtein distance as our metric Δ.

$$C = \{s \in \text{Paths} : \Delta(s, \Pi) \leq d\} \tag{6}$$

An ICFG represents loops with backedges. Thus, a concrete path that iterates in a loop is not a simple path in an ICFG. To statically extract paths and compute their distance, we eliminate all backedges by infinitely unrolling all loops to form an infinitely unrolled ICFG, denoted ICFG$_\infty$. Figure 4 shows a loop in an ICFG and its unrolled



(a) CFG with loop (b) ICFG$_\infty$

Figure 4: A CFG and its equivalent ICFG$_\infty$.

Algorithm 1 Generate paths Levenshtein distance d from Π

```
Input: $d$ // distance
Input: ICFG
Input: $\Pi$
  $G$ = reverseArcs(ICFG)
  $\Pi^r$ = reverse($\Pi$)
  result-paths ← $\emptyset$
  q.enqueue($\langle \Pi^r[0], \langle\rangle \rangle$) // vertex, path
  while not q.empty? do
    $\langle v, p \rangle$ = q.dequeue()
    if $v = \Pi^r[-1] \wedge l(p, \Pi^r) \leq d$ then
      result-paths ← result-paths ∪ $\{p\}$
    end if
    $e = \min(|p| - 1, |\Pi|)$
    if $l(p, \Pi^r[0, e]) \leq 2d$ then
      p.append($v$)
      q.enqueue($\langle n, p \rangle$), $\forall n \in \eta_G(v)$)
    end if
  end while
  return result-paths
```

representation in an ICFG$_\infty$. All executions have simple paths in an ICFG$_\infty$. ICFG$_n$, a finite subgraph of an ICFG$_\infty$, unrolls all loops n times. All terminating programs iterate each loop a finite number of times, and can be captured by an ICFG$_n$. In particular, all paths within distance d of Π_{i_b} statically exist in an ICFG$_n$ for some n.

$$\eta_G : V \to 2^V \tag{7}$$
$$l : \Sigma^* \times \Sigma^* \to \mathbb{N}_0 \tag{8}$$

The function η_G in Equation 7 returns the neighbors of a vertex in the graph G, and l in Equation 8 calculates the Levenshtein distance of two strings. Algorithm 1 uses these functions to calculate C. For the sequence s, Algorithm 1 uses the notation $s[i]$ to denote the i^{th} component of s, $s[-1]$ to denote the last component of s, and $s[i, j]$, for $0 < i < j \leq |s|$ to denote the substring from i to j in s.

Algorithm 1 reverses Π and the arcs in the given ICFG, before traversing each path until it exceeds a distance budget of $2d$. Paths that reach the entry are retained only if they pass the more stringent distance d as they can no longer become closer to Π. Algorithm 1 handles either an ICFG or an ICFG$_\infty$, but is easier to understand when traversing an ICFG$_\infty$.

The set of candidate paths C that Algorithm 1 outputs may contain infeasible paths, such as ones that iterate a loop once more than its upper bound. Such paths are discarded in WP$_d$'s second phase. Each path in C is traversed. When a path reaches a node, a theorem-solver attempts to satisfy its current predicate. A path whose predicate is unsatisfiable is discarded. For example, a path that iterates a loop beyond its upper bound generates a predicate similar to $i = 4 \wedge i < 4$. The weakest preconditions of satisfiable paths are computed and combined to form a disjunction:

$$\mathrm{WP}_d(P, \varphi, \Pi, d) = \bigvee_{c \in C} \mathrm{WP}(c, \varphi). \tag{9}$$

As d increases, WP_d calculates the weakest precondition of a larger subset of the paths over which standard WP computes; in the limit, WP_d becomes WP:

$$\lim_{d \to \infty} \mathrm{WP}_d(P, \varphi, \Pi, d) = \mathrm{WP}(P, \varphi). \tag{10}$$

Under standard WP, the number of paths grows exponentially with the number of jumps in the target program. Polymorphism exacerbates this problem in object-oriented programs. In WP_d, the distance factor d controls the number of paths used in the weakest precondition computation. When $d = 0$, WP_d only computes WP on the path Π. Varying d allows one to trade off the precision of the computed predicate against the efficiency of its computation. In contrast with standard WP, moreover, WP_d selects paths close to the erroneous path Π_{i_b}. Thus, WP_d can, in principle, find a counterexample using fewer resources than standard WP.

Unlike WP, WP_d does not need loop invariants. The difficulty of deriving loop invariants has hampered the application of WP. Indeed, current tools have adopted various heuristics, such as iterating a loop 1.5 times (loop condition twice, its body once), to circumvent the lack of loop invariants [6, 13]. Under WP_d, edit distance determines candidate paths and therefore the number of the loop iterations of each loop along the path. WP_d supports context-sensitivity via cloning; the distance budget determines whether WP_d explores a path with one more or one fewer recursive calls. Though edit distance may construct infeasible paths, WP computation along such a path will produce an unsatisfiable predicate which will cause WP_d to discard the path.

3.3 Detecting Violations of Coverage

With the help of WP_d, the coverage of a fix f, $cov(f)$, can be computed in three main steps, as shown in Equation 11:

$$\overbrace{\exists x_1 \cdots x_n\ [\mathrm{SE}(P_f, \underbrace{\mathrm{WP}_d(P_b, \neg\varphi, \overbrace{e(P_b(i_b))}^{\text{step 1}}, d)}_{\text{step 2}} \wedge \varphi)]}^{\text{step 3}} \tag{11}$$

1. Extract the concrete path Π_{i_b} that the buggy input i_b induces;
2. Compute $\mathrm{WP}_d(P_b, \neg\varphi, \Pi_{i_b}, d) = \alpha$; and
3. Symbolically execute P_f using α to derive P_f's postcondition ψ, and eliminate the non-input variables x_i from the clause $\psi \wedge \varphi$ to yield $cov(f)$.

In the first step, we run $P_b(i_b)$, then apply e to extract Π_{i_b}. Recall that φ is the failing assertion. In the second step, α under-approximates the set predicate of $\tilde{i}_b$, the true bug-triggering input domain. In this step, we compute α for various values of d, depending on resource constraints. The precision of our under-approximation depends on d. Starting from the weakest precondition computed for various d in step 2 anded with the assertion φ, the third step uses symbolic execution [24] to compute the set of bug-triggering inputs handled by the bug fix. We eliminate non-input variables, *i.e.* intermediate and output variables, from the clause, as they are irrelevant to coverage, which concerns only inputs.

Algorithm 2 Compute the disruption of a fix

```
Input: $I$ // The program's input domain
Input: $p_b$ // The buggy version of the program
Input: $p_f$ // The fixed version of the program
Input: $T : I \to O$ // The program's test suite
 1: $R = \emptyset$
 2: $\forall (i, o) \in T$ do // Standard regression testing
 3:     if $p_f(i) \neq o$ then
 4:         $R \leftarrow R \cup \{i\}$
 5:     end if
 6: end for
 7: $\forall i \in$ a random subset of $I$ do
 8:     if $p_b(i) \Rightarrow \varphi \wedge p_b(i) \neq p_f(i)$ then
 9:         $R \leftarrow R \cup \{i\}$
10:     end if
11: end for
12: return $R$
```

To decide whether f is a bad fix in terms of coverage, we check the validity of $\psi \to \varphi$. If it is valid, the fix does not violate the coverage requirement. Otherwise, we deem f a bad fix and report any counterexamples to the validity of $\psi \to \varphi$ as new bug-triggering inputs. Our assertion that f does not cover $\tilde{i}_b$ is sound because α under-approximates $\tilde{i}_b$, *i.e.*, $\{i \in I : \alpha\} \subseteq \tilde{i}_b$.

3.4 Detecting Violations of Disruption

To measure the disruption of the fix f, we first run P_f on the test suite, as is conventional, because test suites are crafted to exercise important execution paths [16, 32]. Each test failure is a disruption. Given a specification of a program's input I, we then randomly choose an input $i \in I \setminus \tilde{i}_b$. We compare the output of P_b to P_f to find errors not anticipated by the test suite. Each time $P_b(i) \neq P_f(i)$, we have found another disruption. Because $\hat{i}_b$ under-approximates the actual bug-triggering input domain $\tilde{i}_b$, we ignore inputs that trigger $\neg\varphi$ in P_b.

Algorithm 2 computes the disruptions of a fix, returning a set of failing inputs. The loop at lines 7–11 samples inputs from I, ignoring those that trigger the original bug. We use the fact that P_b fails the assertion φ to discover such inputs. In comparing P_f and P_b on those inputs, we do not assume that P_b has only the one bug b and works correctly for all other inputs; instead, we simply assume that we can consider each bug in isolation. Further, outside of $\tilde{i},_b$ P_b usefully approximates the ideal behavior of P; when P_b is a release, deployed version of a program, it presumably passed regression testing.

4. EMPIRICAL EVALUATION

Our evaluation objective is two-fold: to demonstrate the feasibility and utility of our approach, and to differentiate WP_d from WP. First, we describe our implementation, our computing environment and how we selected our test suite. We then show that FIXATION detects the bad fix in our motivating example, as well as five others. We compare WP_d to WP by showing how WP_d accumulates path predicates, and thus subsets of the true bug-triggering input domain $\tilde{i}_b$, as a function of its parameter d. Finally, we close by describing how a developer might use FIXATION to discover bad fixes and instead commit good ones.

4.1 Implementation

Many tools and techniques exist for detecting the disruption of a fix, so we focused our implementation on determining fix coverage. We used the WALA framework [20] to extract the concrete buggy path induced by a bug-triggering input and build a CFG, from which we extract candidate paths using Algorithm 1. The extracted concrete path is the sequence of basic blocks traversed during a particular execution of the program. ICFG$_n$ is produced by traversing the CFG of each method and unrolling each loop the number of times it iterated in the concrete path and an additional x times, according to an unrolling parameter x; thus, $n = x + y$, where y is a loop that iterated the most times in the concrete path. The unrolling parameter allows FIXATION to explore paths that loop more often than the concrete path specifies. We then run our implementation of Algorithm 1 over this ICFG$_n$ and feed each resulting path predicate to the SMT-solver CVC3 [3], keeping only those that are satisfiable.

Java PathFinder is the Swiss army knife of Java verification [37]. FIXATION uses JPF's Symbc component to symbolically execute the fixed program, using the weakest precondition produced above as the precondition. If, given this precondition, the assertion fails in the fixed program, Symbc generates a concrete input that causes the failure and returns it as a counterexample.

4.2 Experimental Setup

We built and ran our evaluations on a Dell XPS 630i with 2.4GHz QuadCPU processors and 3.2 GB of Memory, running Ubuntu 8.04 with kernel Linux2.6.24-21-generic. FIXATION is built for and runs on JRE 6.

Although we found many bad fixes in the three projects we explored, most of them were not fit for evaluation: either the bug comments were unclear or no fix was uploaded. No convention appears to govern the use of Bugzilla. Some programmers tend to write detailed logs of their fix activity and upload their fix, but most do not. We selected the first six bugs we found whose comments proved the existence of bad fix and that had an attached fix. Five of the bad fixes are from Rhino and `MultiTask` is from Ant[3].

Currently, FIXATION does not directly work on the original, unmodified code, since both WP$_d$ and Symbc (JPF v4.1) support only Java programs consisting of statements and expressions that use only boolean and integer variables. Thus, we first manually sliced away all code not related to the bugs that caused the bad fixes. We then transformed the code into an integer program that, given the same inputs, traverses the same paths as the sliced version of the original program. Since FIXATION simply ignores non-integer language constructs, like function calls and field or array accesses, we left them in place to approximate the original program as closely as possible. We manually transformed the conditionals in the original code into integer conditionals. To rewrite `fun instanceof NoSuchMethodCall`, we introduced the integer variable `fun_int` and the integer constant `NoSuchMethodCall_int`, then replaced the original conditional with the conditional `fun_int == NoSuchMethodCall_int`. We added logic as necessary so that `fun_int` correctly tracked the type of original object `fun`. At each point the bug manifested itself in the original program, we added an assertion.

Our principal goal was to faithfully retain the inherent complexity of the program's logic through the transformation. Table 1 presents evidence that we succeeded; it shows the raw lines of code, the nodes and arcs in the ICFG, and the Cyclomatic complexity (CC) of ICFGs before (P) and after P_i transformation. For a program with one exit point, Cyclomatic complexity equals the number of decision points in the program plus one [26]; we increase it in all six cases.

4.3 Experimental Results

[3] The sliced and transformed versions of the bad fixes are available at http://wwwcsif.cs.ucdavis.edu/~gu/bugs.htm.

Name	Loc		Nodes		Arcs		CC	
	P	P_i	P	P_i	P	P_i	P	P_i
`NoSuchMethod`	60	65	60	64	70	75	12	13
`MultiTask`	23	36	51	62	54	68	5	8
`Substring`	8	17	10	16	10	18	2	4
`NativeErr`	9	20	10	18	10	21	2	5
`Loop`	34	42	42	46	46	51	6	7
`PathExp`	114	133	103	119	124	147	23	30

Table 1: Bad fix CFG complexity.

```
instructionCounting++;
...
stackTop -= 1 + indexReg;
if( fun == InterpretedFun ) {
  return processInterFun();
}
if( fun == Continuation ) {
  return processCon();
}
if( fun == IdFunctionObject ) {
  return processIdFunObj();
}
...
assert( false ); // Should never execute.
```

Figure 5: Sliced integer version of `NoSuchMethod`.

Table 2 details the results of evaluating the six examples. The second column briefly describes each bad fix. The third column contains the counterexample FIXATION reports. The fourth column d presents the edit distance used to construct the candidate paths. C denotes the number of paths FIXATION explored before determining the fix to be bad. As a point of reference for C, P_a is the total number of paths. Time records the time-to-completion.

Neglected Execution Paths The first four bad fixes in Table 2 are all due to a programmer's ignorance of potential buggy paths. FIXATION detected the buggy paths missed by each of these bad fixes while exploring a limited number of paths. Once it detected a bad fix, FIXATION reported a counterexample to help programmers refine their fixes. Since the bad fixes `MultiTask`, `Substring` and `NativeErr` all exhibit essentially the same symptoms as `NoSuchMethod`, we describe only `NoSuchMethod`.

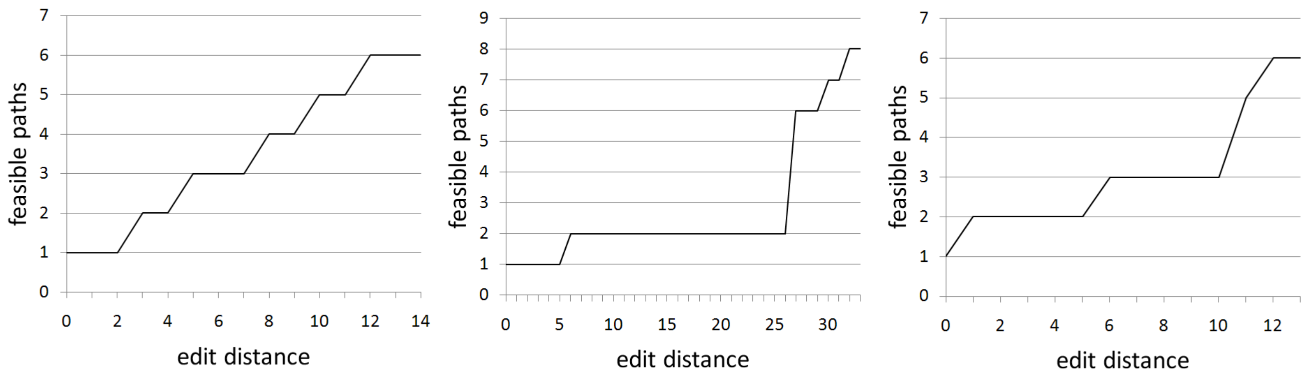
Figure 5 lists the original buggy code that required three fixes as chronicled in Section 2. The original bug-triggering input `fun == NoSuchMethodShim && op != Icode_TAIL_CALL` reminded the programmer that there was no block for `NoSuchMethodShim`, so the programmer committed the first fix in Figure 1. FIXATION took the initial bug-triggering input and buggy code, ran with distance d set to zero, and discovered the bug-triggering input domain `fun != InterpretedFun && fun != Continuation && fun != IdFunctionObject`. FIXATION then symbolically executed the first fixed program, imposing that predicate together with the set predicate of the program's input domain as the initial precondition[4]. Given that precondition, FIXATION reached and implied an assertion failure, then reported the fix bad and returned the counterexample `fun == NoSuchMethodShim && op == Icode_TAIL_CALL`.

A Bad Fix in a Loop Standard WP needs loop invariants, which are difficult to derive in general. Current tools usually adopt heuris-

[4]Without the predicate of the program's input domain, Symbc may generate nonsensical inputs. Assume that a program's input domain is $x > 0 \land x < 10$ and the computed weakest precondition is $x > 5$. With only $x > 5$, Symbc could generate the illegal input $x = 20$.

Name	Description	Counterexample	d	C	P_a	Time (s)
`NoSuchMethod`	Neglect an input.	`fun == NoSuchMethodShim && op == Icode_TAIL_CALL`	0	1	30	0.664
`MultiTask`	Forget targets' size can be zero.	`type == VECTOR && size == 0`	3	32	48	1.637
`Substring`	Miss a condition.	`i == 1 && sT == SUB_NULL`	2	3	3	0.598
`NativeErr`	Miss handling an exception.	`type == NativeError`	2	4	5	0.938
`Loop`	Fail to detect bugs in loop.	`i_c == 6 && m_l = 3`	5	354	∞	42.542
`PathExp`	Miss bugs on different paths.	`tG == -10000 && cT == T_PRI && cI == T_NULL && op == T_SHNE && sC == T_TRUE`	27	234	1021	8.308
`PathExplosion2`	Miss bugs on different paths.	`tG == -10000 && cT == T_PRI && cI == T_NULL && op == T_SHEQ && sC == T_FALSE && ···`	6	237	1021	8.518

Table 2: FIXATION results.



(a) `Loop` (b) `PathExp 1` (c) `PathExp 2`

Figure 6: Feasible paths as a function of edit distance.

```
private static final int M_T = 2;
private static final int I_L = 4;
...
public AdapterClass createAdapterClass(
      String adapterName, Scriptable ins[],
      int adapterType, int i_c, int m_l) {
   ...
   assert( i_c <= I_L );
   ...
   for( int i = 0; i < i_c; i++ ) {
      assert( i <= I_L && m_l <= M_T );
      ...
   }
   ...
}
```

Figure 7: Sliced integer version of `Loop`.

tics at the cost of sacrificing precision [6, 13]. Here, we show how WP$_d$ tackles the loop problem. Two bugs lurk in Figure 7 — one outside the loop and the other inside. The initial bug-triggering input `i_c == 5 && m_l == 2` exposes only the bug outside the loop. The first committed fix resolved only this bug. We ran FIXATION with $d = 5$ on this first fix. FIXATION explored 354 candidate paths, most of which were unsatisfiable. For example, computing WP on the path that takes the true branch inside the for loop in its 4th iteration generates the unsatisfiable clause `i==3 && i>4`. Disjuncting predicates from feasible paths, FIXATION reported the bug-triggering input domain `(i_c == 5 && m_l > 2) || (i_c == 5 && m_l <= 2) || (i_c == 6 && m_l > 2) || (i_c == 6 && m_l <= 2)`. Given this predicate as its precondition, FIXATION output the counterexample `i_c == 6 && m_l == 3`.

In essence, WP$_d$ is unaware of loops. The candidate paths over which WP$_d$ computes the weakest precondition are all simple paths, drawn from an ICFG$_n$. Figure 6a shows the number of feasible path predicates WP$_d$ discovered as a function of the edit distance. It illustrates that WP$_d$ can produce useful results when given limited resources. `Loop` has infinite paths and therefore infinite feasible paths, which we cut off at $d = 14$. By way of comparison, we ran this example using ESC/Java2 and JPF Symbc, without our path restriction. Using its default settings[5], ESC/Java2 failed to report a potential postcondition violation. Given the domain restriction we inferred in our first phase, our backward symbolic execution from the assertion failure, Symbc does reach the assertion failure, but at the cost exploring all loop iterations up to the failure, as opposed to only those iterations within d of the failing iteration; more importantly, Symbc continued searching until it reached loop iteration 1,000, when we killed it.

Path Explosion WP$_d$ balances scalability and coverage via its edit distance parameter d. `PathExp` illustrates how FIXATION detects potential bugs even when considering subsets of the potentially feasible paths. `PathExp` occurred because Rhino failed to ensure `(undefined === null)` evaluated to `false` as required by the ECMA (JavaScript) specification. In Figure 8, six nested bugs

[5]ESC/Java2 does not support assertions, so we translated each assertion into a postcondition annotation.

```
if ( tG == -1 ) {
  if ( c_T == T_PRI && c_I == T_NULL ) {
    //bug 1
    assert( op!=T_SHEQ && op!=T_SHNE );
    ... //assignments to op
    //bug 2
    assert( op!=T_SHEQ && op!=T_SHNE );
    ...
  }
} else {
  if ( c_T == T_PRI && c_I == T_NULL ) {
    ...
    //bug 3
    assert( op!=T_SHEQ && op!=T_SHNE );
    ... //assignments to op
    //bug 4
    assert( op!=T_SHEQ && op!=T_SHNE );
    ...
    if ( op == T_EQ || op == T_SHEQ ) {
      if ( sC==T_FALSE ) {
        markLabel(popGOTO, popStack);
        addByteCode(ByteCode_POP);
        //bug 5
        assert(op!=T_SHEQ && op!=T_SHNE);
      }
      ...
      //bug 6
      assert( op!=T_SHEQ && op!=T_SHNE );
    }
  }
}
```

Figure 8: Sliced integer version of `PathExplosion`.

are distributed in different paths. From the initial bug-triggering input, the programmer discovered and fixed only bugs 1 and 2. We ran FIXATION on `PathExp`, gradually increasing d as shown in Figure 6b. As the figure depicts, WP$_d$ finds no additional paths until $d = 27$. Exploring Figure 8, we find that bugs 3–6 are all in the `else` block of the first `if` statement, with conditional `( tG == -1 )`. A path that traverses any of bugs 3–6 shares few nodes with the original buggy path, which traverses bugs 1 and 2. At $d = 27$, FIXATION finds the new feasible predicate `tG != -1 && c_T == T_PRI && c_I == T_NULL && op == T_SHNE && sC == T_TRUE` after exploring 234 paths. Armed with this predicate, FIXATION can symbolically execute the first fixed program and report a counterexample exposing bugs 3 and 4.

Perhaps, as usual, the programmer was harried and rushing. In any case, he committed a fix that corrected only bugs 1 and 2. Had the programmer run FIXATION, he would have been aware of the counterexample that his fix failed to handle. Figure 6c shows the result of running FIXATION with increasing d on the partially fixed program. At $d = 5$, FIXATION detects two new feasible weakest preconditions after exploring 237 paths. These predicates generate two counterexamples that identify the remaining two bugs. WP$_d$ offers a flexible, principled, and resource-judicious way to search paths near a bug-triggering path.

4.4 Threats to Validity

FIXATION's edit distance parameter d allows its user to trade-off performance against the completeness of FIXATION's approximation of the bug-triggering input domain. Determining the optimal setting of d to obtain a better result is an interesting problem. The paths WP$_d$ traverses depends on d as well as the structure of the CFG. Gradually increasing d until detecting interesting paths or exceeding a resource threshold, such as time or number of paths explored, appears to be a good heuristic.

FIXATION is currently not optimized. Caching the predicates of already explored paths would avoid redundant computation. Tabulating function summaries for reuse can also cache WP computations. Adopting lightweight predicate on-the-fly feasibility checking, à la Snugglebug [6], might remove infeasible paths at an earlier stage.

We model bugs as assertion failures. Thus, FIXATION's applicability depends on assertions being available, inferred, or written by a programmer. In many cases, obtaining such an assertion is trivial (as in thrown, fatal, exceptions). In others, it can be difficult and is an external threat to the validity of our approach: an empirical study to investigate and classify those cases would help users know when FIXATION is and is not practical. As with any manual study, our results may exhibit selection bias. A larger empirical study would also help in showing the technique generalizes.

We inherit our limitation to integer programs from our symbolic execution components. This restriction makes the values of d we report optimistic as we operate on slightly smaller, sliced, versions of the original program. Thus, our reliance on slicing is both a construct and, because slicing limits the applicability and scalability of our approach, an external, threat to validity. As the state-of-the-art in symbolic execution improves (e.g. via techniques such as delta-execution [10]), so will the applicability of our approach. Nonetheless, the evaluation results are encouraging. FIXATION detects bad fixes and reports counterexamples that can help programmers realize the limitation of particular fix. WP$_d$ has shown itself to be a promising approach to managing the path-explosion problem and side-stepping the loop invariant problems.

5. RELATED WORK

Our work is the first to offer a systematic methodology for assessing the quality of a bug fix. It is related to the large body of work on software testing and analysis. This section summarizes the most closely related efforts. We divide related work into three categories: practical computation of weakest precondition, automatic test input generation, and studies of bug fixes and code changes.

5.1 Practical Computation of WP

Dijkstra's weakest precondition [11] has been extended to check the correctness of object-oriented programs. ESC/Java pioneered its use in Java [13]. ESC/Java requires user-defined annotations to specify the precondition, postcondition and invariants. By checking the validity of the verification condition generated from guarded commands, ESC/Java warns of potential bugs such as postcondition violations or null pointer dereferences. ESC/Java's checking is modular so it relies on user annotation for procedure calls. To handle loops, it heuristically iterates 1.5 times.

Snugglebug [6] presents an interprocedural WP technique. It introduces directed call graph construction, generalization, and a current search heuristic to improve the performance and precision. Polymorphism means that Java programs face the dynamic-dispatch problem when encountering function calls; directed call graph construction helps find the exact callee without exhaustive search. Generalization enhances function summary reuse using tabulation. The current search heuristic of Snugglebug prioritizes paths with less looping or call depth.

Path-based WP computation complements WP computation over an entire program. Applying counterexample-driven refinement, BLAST [19] and SLAM [2] check an abstract path to see if it corresponds to a concrete trace of the program reaching an error state. He and Gupta proposed path-based weakest precondition to locate and correct an erroneous statement in a function [17]. Their path-based weakest precondition is similar to our WP$_d$ when $d = 0$. We have assumed a fix at least covers the original bug-triggering input, so single-path approaches may not handle the bad fix problem. Our approach generalizes path-based WP by parameterizing the distance budget d and allowing arbitrary non-zero distances.

Our WP$_d$ filters the paths over which standard WP computes. Instead of computing WP on all paths, WP$_d$ approximates the true bug-triggering input domain by computing WP on paths that are close to the original buggy path. Users control the performance and coverage of WP$_d$ through its edit distance parameter. As d increases, WP$_d$ generates more paths and takes more time to compute. Another insight of WP$_d$ is that computing the weakest precondition of simple paths near a known concrete path is precise and does not rely on heuristics: the concrete path specifies how to resolve a dynamic dispatch target or determine how often to traverse a loop.

5.2 Automatic Test Input Generation

Automatic test input generation is an active area of research. Dozens of techniques and tools have been proposed. For brevity, we highlight only some of the closely related work.

CUTE [35] and DART [14] combine concrete and symbolic execution to generate test inputs for C programs. Java PathFinder (JPF) [37] performs generalized symbolic execution to generate inputs for Java programs. We use Symbc from JPF [30] to generate counterexamples. We impose preconditions to guide the symbolic execution: Given the set predicate of an approximation of the bug-triggering input domain as the precondition, we ask whether the fixed program can reach the assertion failure or not. This precondition restricts the search space and enhances the performance of Symbc. If Symbc outputs a concrete input that causes the assertion to fail, we deem the fix bad and return that input as a counterexample. Csallner *et al.*'s work [8] combines static checking and concrete test-case generation. They perform random testing using the counterexample predicate generated by ESC/Java. Random testing may miss some paths. We symbolically execute all paths under imposed precondition. Beyer *et al.* extend BLAST to generate testcases by finding inputs that satisfy predicates collected from all paths in the program [4]. We also collect predicates to generate inputs, but for a different purpose: Beyer *et al.* seek to exhaustively test one program, while we use symbolic execution on a buggy and a fixed version of a program to evaluate fixes.

5.3 Bug Fixes and Code Changes

Research on bug fixes mainly falls into two camps: mining software repositories and empirical study. BugMem [22] mines bug fix history to predict potential bugs. Kim *et al.* predict faults by consulting bug and fix caches they build [23]. Their work shows that bugs exhibit locality and inspired our design of WP$_d$. Anvik *et al.* applied machine learning to find programmers who should be responsible for the fix [1]. Weiss *et al.* predict fixing effort needed for a particular bug [38] and Śliwerski *et al.* proposed a way to locate code fixed using repository mining [36]. Most of the work in this domain is probabilistic and may not be precise. We propose a sound analysis for evaluating bug fixes: every bad fix we detect is an *actual* bad fix.

Code change is fundamental to software development. Ryder and Tip propose change impact analysis to find failure inducing changes and thus judge the quality of change [33, 39]. We consider a subset of the changes they consider, specifically bug fixes. Change impact analysis applies delta debugging on a sequence of atomic changes drawn from the comparison of two versions to locate suspicious changes, while we combine our distance-bounded weakest precondition with symbolic execution to judge the quality of a fix. Differential symbolic execution [29] seeks to precisely characterize the differences between two program versions. DSE exploits abstract summaries of code that is common between two program versions, as the effects of such code do not contribute to differences. In contrast, FIXATION seeks to identify not only issues of disruption, but also coverage: a subset of inputs for which both versions behave the same, *viz.* by manifesting a particular bug.

McCamant and Ernst compare operational abstractions of components and their potential replacements to predict the safety of a component upgrade [27]. Our approach is more fine-grained: assertions need not refer to the modified component as operational abstractions must. Their focus is different from ours: they seek to verify that a newer component will behave as expected under the conditions its predecessor was exposed to, and we seek to verify that a component that does indeed behave differently (due to a bug fix) does so safely (*i.e.* without disruption) and completely (*i.e.* handling all of $\tilde{i}_b$).

Regression testing validates modified software to ensure changed code has not adversely affected unchanged code [16, 28, 32, 34]. The cost of regression testing has been extensively studied and shown that test suite size can be reduced without compromising safety [15, 31]. Currently, we combine regression and random testing to check the disruption of a fix.

6. CONCLUSION AND FUTURE WORK

When run on buggy and allegedly fixed versions of a program, FIXATION reports a new bug-triggering input drawn from an under-approximation of the true bug-triggering input domain. This new bug-triggering input is a counterexample to the implicit assertion that the fix is good. FIXATION can miss bad fixes if it fails to explore a buggy path, but it is sound when it asserts that a fix is bad: every counterexample FIXATION reports is certain to cause the fixed program fail the assertion.

We have introduced the bad fix problem and provided empirical evidence of its existence in real projects. We have formalized the bad fix problem and proposed an approach that combined our distance-bounded weakest precondition with symbolic execution to evaluate fixes and detect bad ones. We implemented our idea in a prototype FIXATION and evaluated it: FIXATION was able to detect bad fixes extracted from real-world programs.

In the future, we plan to extend FIXATION to support more language features in Java and make it applicable to real code. We intend to implement the optimizations mentioned. Unit testing is an immediate application of FIXATION. A failed testcase is an ideal original bug-triggering input for FIXATION. Self-contained assertions in a test suite will allow FIXATION to work directly on the code and obviate manually constructing and inserting the assertion. The fact that the distribution of code tested by unit testing is relatively local is likely to be a good fit for WP$_d$.

Acknowledgments

We would like to thank Radu Grigore and Joseph R. Kiniry for helpful discussions about ESC/Java2, Stephen Fink and Manu Sridharan for generously answering our questions about WALA, and Corina S. Păsăreanu for promptly and concisely explaining how to run Symbc of Java Pathfinder. We appreciate Christian Bird's help during our empirical exploration on Bugzilla databases. We are also grateful to the anonymous ICSE reviewers for their valuable comments.

References

[1] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, 2006.

[2] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2002.

[3] C. Barrett and C. Tinelli. CVC3. In *CAV '07: Proceedings of the* 19*th* *International Conference on Computer Aided Verification*, July 2007.

[4] D. Beyer, A. J. Chlipala, and R. Majumdar. Generating tests from counterexamples. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, 2004.

[5] Bugzilla. http://www.bugzilla.org/.

[6] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, volume 44, 2009.

[7] J. S. Collofello and S. N. Woodfield. Evaluating the effectiveness of reliability-assurance techniques. *Journal of Systems and Software*, 9(3), 1989.

[8] C. Csallner and Y. Smaragdakis. Check 'n' Crash: combining static checking and testing. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, 2005.

[9] C. Csallner, Y. Smaragdakis, and T. Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Transactions Software Engineering Methodology*, 17(2), 2008.

[10] M. d'Amorim, S. Lauterburg, and D. Marinov. Delta execution for efficient state-space exploration of object-oriented programs. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, 2007.

[11] E. W. Dijkstra. *A Discipline of Programming*. October 1976.

[12] J. Dolby, M. Vaziri, and F. Tip. Finding bugs efficiently with a SAT solver. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007.

[13] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002.

[14] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[15] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Transactions Software Engineering Methodology*, 10(2), 2001.

[16] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *OOPSLA '2001: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2001.

[17] H. He and N. Gupta. Automated debugging using path-based weakest preconditions. In *FASE '04: Proceedings of the Fundamental Approaches to Software Engineering, 7th International Conference*, volume 2984 of *Lecture Notes in Computer Science*, 2004.

[18] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *SPIN '2003: Proceedings of the Tenth International Workshop on Model Checking of Software, volume 2648 of Lecture Notes in Computer Science*, 2003.

[19] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2002.

[20] IBM. T.J. Watson libraries for analysis. http://wala.sf.net.

[21] IDC. A Telecom and Networks market intelligence firm. http://www.idc.com.

[22] S. Kim, K. Pan, and E. E. J. Whitehead, Jr. Memories of bug fixes. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006.

[23] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, 2007.

[24] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

[25] V. Levenshtein. *Binary codes capable of correcting deletions, insertions and reversals (in Russian)*. 1965.

[26] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4), 1976.

[27] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, 2003.

[28] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. *SIGSOFT Software Engineering Notes*, 29(6), 2004.

[29] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008.

[30] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, 2008.

[31] G. Rothermel and M. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8), Aug 1996.

[32] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions Software Engineering Methodology*, 6(2), 1997.

[33] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2001.

[34] R. Santelices, P. Chittimalli, T. Apiwattanapong, A. Orso, and M. Harrold. Test-suite augmentation for evolving software. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, 2008.

[35] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, 2005.

[36] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR '2005: International Workshop on Mining Software Repositories*, 2005.

[37] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, 2004.

[38] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. Predicting effort to fix software bugs. In *Proceedings of the 9th Workshop Software Reengineering*, May 2007.

[39] J. Wloka, E. Hoest, and B. G. Ryder. Tool support for change-centric test development. *IEEE Software*, 99(PrePrints), 2009.